AD-A102 477   OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
AN APPROACH TO RELIABLE INTEGRATION TESTING.(U)
MAY 81 A HALEY, S ZWEBEN   F49620-79-C-0152
UNCLASSIFIED OSU-CISRC-TR-81-5   AFOSR-TR-81-0578   NL

1 OF 1
AD
A102477

END
DATE
FILMED
9-81
DTIC

1-0578

TECHNICAL REPORT SERIES

LEVEL II

3

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

18 AFOSR

19 TR-81-0572

3

14 OSU-CISRC-TR-81-5

6 AN APPROACH TO RELIABLE INTEGRATION TESTING

by

10 Allen Haley ~~and~~ Stuart Zweben

11 Interim

12 43

DTIC ELECTE AUG 5 1981

Research supported in part by

Air Force Office of Scientific Research
15 F49620-79C-0152

~~National Science~~ Foundation
NSF-MCS-8018769

16 2304

17 A2

Computer and Information Science Research Center
The Ohio State University
Columbus, OH 43210

May, 1981

401516

ABSTRACT

A testing strategy which involves integrating a previously validated module into a software system is described. It is shown that, when doing the integration testing, it is not enough to treat the previously validated module as a "black box", for otherwise certain integration errors may go undetected. For example, an error in the calling program may cause an error in the module's input which will only result in an error in the module's output along certain paths through the module. The results indicate that such errors can be detected by the module by retesting a set of paths whose cardinality depends only on the dimensionality of the module's input space, rather than on the module's path complexity.

PREFACE

This report is the result of research supported in part by the Air Force Office of Scientific Research under contract F49620-79c-0152 and by the National Science Foundation under grant MCS-8018769. It is being published by the Computer and Information Science Research Center (CISRC) of the Ohio State University in conjunction with the Department of Computer and Information Science. CISRC is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories.

INDEX TERMS

Software Engineering, Computer Program Testing, Integration, Modules

TABLE OF CONTENTS

AN APPROACH TO RELIABLE INTEGRATION TESTING

Allen Haley

Stuart Zweben

1. Introduction

While program testing remains the most widely used method of validating computer software, the computer field is still in the unenviable position of lacking practical, effective testing methodologies. Test data chosen by random or ad hoc methods may provide at least some method of testing but indicates little as to how well tested is the resulting software. Practical testing strategies which attempt to satisfy certain necessary conditions (such as statement or decision coverage approaches) can be shown incapable of detecting many errors (see, e.g. [Howden, 76, Westley, 79]). Finally, strategies which attempt to guarantee detection of wide classes of errors (e.g., path oriented strategies) require too much testing to be of practical value especially where large programs are involved. In such cases one must look for ways to reduce the amount of testing required without placing severe penalties on the amount of confidence in the correctness of the tested program.

One possible approach to achieving this reduction is motivated by considering the problem of program development. In developing the solution to a large, complex problem, it is customary to form subdivisions which abstract interesting aspects of the total solution. These subdivisions might then be refined, implemented, and tested as independent units of the total system and then integrated to form a complete working solution to the original problem. When viewing the integrated program as the object to be tested, it may well be the case that the complexities are too great to make certain testing strategies practical. For example, consider a program P consisting of subprogram P1 containing m paths followed by subprogram P2 containing n paths. The integrated program can have a total of m * n paths (see Figure 1 for m=3 and n=4), since any of the m paths in P1 can be followed by any of the n paths in P2 .

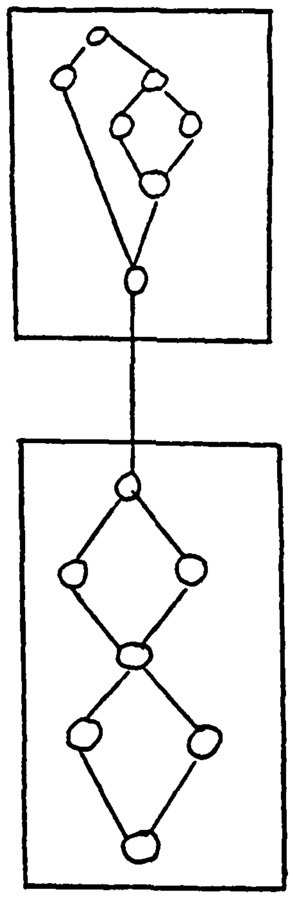


Figure 1

Integration of Subprograms with 3 and 4 paths, respectively

In the course of developing P however, it may well be the case that both P1 and P2 have been tested separately. It would be desirable if the correctness information obtained in unit testing P1 and P2 could be used in validating P . If the individual modules[1] do not contain a large number of paths, it may in fact be possible to test all possible paths in each module. If the additional testing required at integration time was negligible compared to the unit testing overhead (for example, if we could ignore the internal control structure of a tested module when integrating it), the result would be a reduction of the magnitude of the testing problem from $O(m*n)$ to $O(m+n)$. While this represents in some sense an ideal situation, it is clear that with such a potential for complexity reduction, even a less than ideal solution might represent a considerable improvement and yet provide a substantial degree of practicality.

Research to date in program testing has concentrated on a single program unit. There is nothing inherently wrong or undesirable with this approach, since in order to be able to say anything about a large problem, it makes sense

---

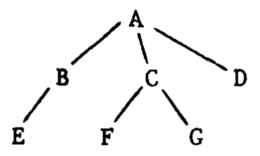
1. The notion of a module has been characterized in many different ways, and several authors have proposed criteria for what constitutes a "good module" [see, e.g. Parnas, 72, Yourdon, 79, Stevens, 74]. It will suit our purposes to allow a module to be any single entry, single exit block of code which can contain an arbitrary amount of internal control structure. For simplicity, we may represent modules as subroutines in this paper, with the understanding that the ideas presented herein are not meant to be restricted to this form of modular structure.

to gain an understanding of the (smaller) units of which the problem is composed. However, given that these individual units must eventually work together as a system, we must be conscious of potential problems that might be encountered at integration time, and develop testing strategies which are sensitive to these problems as well as those of a single program unit.

Thus, the justification for the development of a method of integrating independently tested modules into a single program is (1) to reduce the total testing complexity, and (2) to make the testing procedure conform to the way programs are developed.

2. Integration Testing Philosophies

There are basically two approaches to testing a set of modules which form the overall structure of a system -- top down and bottom up.



System Configuration

1	{E}	5	{C,F,G}
2	{F}	6	{D}
3	{G}	7	{A,B,C,D,E,F,G}
4	{B,E}		

Possible Test Sequence

Figure 2

In bottom up testing, individual modules are first tested in isolation from one another using their own sets of test data. When groups of related modules have been validated, they are integrated into a higher level unit (subsystem) which is then tested. The subsystem tests in general require new test data, since different inputs and outputs are involved at the higher level. Larger and larger subsystems are combined until eventually the entire system is tested as a unit. Using the "system" of Figure 2, a possible sequence of tests using a bottom up technique would be: (1-7). The primary disadvantages of this form of testing are the variety of test data required at each level and the increasing complexity of the subsystems as the integration proceeds.

Top down testing, on the other hand, involves starting with the highest level component of the system and proceeding to the next lower level, etc. Using Figure 2 as a model once again, the "higher level" test should ideally determine that A functions correctly, knowing only the abstract purposes of B, C and D, so that when B, C and D are eventually implemented, it is necessary only to show that they achieve their abstract purpose (that is, theoretically no integration testing is required). However, our ability to select appropriate tests of A based on these abstractions and to certify the correctness of A's output on these tests (which, after all, requires output from abstract, as yet unwritten modules), is quite limited. We are much more likely when testing A to first write stubs for B, C and D which do nothing more than indicate that the second level modules have in fact been called, and later embellish B, C and D so that they can produce correct output for some very small, well known class of possible inputs. While this method helps identify the appropriateness of the invocation of the lower level modules and can speed up the completion of a preliminary version of a complex system, it tends to mask the subtle interrelationships between the components until all are completely developed and an attempt is made to have them work as a unit.

Therefore, one can say that even if a top down testing philosophy is attempted, integration testing will be necessary after the lowest level modules have been completed. That is, some mixture of top down and bottom up testing is probable.

In the remainder of the paper, we will explore the issues involved when a "correct" module (one which produces the appropriate output for any valid input) is integrated into a larger program context, with the goal of identifying testing strategies which are sensitive to integration time errors.

3. Integration Time Errors

In order to be able to characterize the effectiveness of any testing approach, it is necessary to identify those errors which are of interest to the strategy under consideration. Any finite testing procedure is known to be faced with certain inherent limitations as to the errors it is capable of detecting. One of these limitations can be characterized as the "coincidental correctness" problem, whereby the program under examination happens to produce the same results as the (different) desired program on the set of data tested. Thus, a statement such as X=X+2 cannot be differentiated from X=X*2 if the only test data chosen result in X=2 on entry to the statement. Another inherent limitation of any finite strategy has been called the "missing path" problem. This problem arises, for example, when some "special case" has not been appropriately dealt with. Thus there may be some special action to be taken only "IF X=1", which the programmer forgot to include. If none of the test data happen to set this condition, the missing action will not be detected.

Admitting that errors due to coincidental correctness and missing paths may go undetected, the next problem is to try to classify those kinds of errors that we might hope to detect. One proposal, due to Howden [Howden, 76], distinguishes between domain errors and computation errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program. A computation error exists when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. This classification scheme has been used successfully by researchers of the Domain Testing Strategy [White, 80]. The Domain Testing Strategy is designed to detect domain errors, though it also has some ability to detect computation errors.

The notion of domain and computation errors turns out to be useful in characterizing certain types of integration problems. For example, consider a module M which has been thoroughly validated, say by some "Hypothetical Testing Strategy", so that it is free of both domain and computation errors. Module M is to be integrated into a program P. Assume that P has some computation whose result (call it C) is used in some predicate of M but is not used anywhere else in the program (see Figure 3).

```
     READ Ip                                  .
     C = Ip                                   .
P    CALL M ( ..., C,...Om)         M         .
     Op = Om                                  IF C < 4
     PRINT Op                                 THEN Om=1
                                              ELSE Om=2
```

Figure 3

Program Containing a Computation Used Only in a Predicate of a Previously Tested Module

Now suppose that the correct computation in P should have set C to Ip+1 . In validating M , we may have ensured that M produces the correct output no matter which branch of the IF statement is taken, but P will still produce the wrong output if the initial value Ip is such that $3 \leq Ip < 4$. However, if we do not happen to choose a value of Ip in this range we will not catch the error in the computation statement. Notice that, from the point of view of the program P , there is only one path to consider (Read Ip; C=Ip; CALL M (...); Op=Om; PRINT Op) if we ignore the control structure of the module M . Ideally, we would like to be able to ignore the internal structure of M at integration time and deal only with P's structure. Yet this example shows that we must do more than just select a couple of values of Ip and examine the resulting values of Op . In this case, if we were to analyze the integrated program including the module's control structure, we would notice that the program contains a domain error, since values of Ip in the range $3 \leq Ip < 4$ follow the wrong path.

Computation errors cause another problem in ignoring the validated module's control structure at integration time. Assume that the program contains an incorrect computation whose result is passed to the validated module. Further assume that the only use of this result is by some computation in the validated module. As an example, suppose P is the same as in Figure 3, but M is changed as in Figure 4.

$$\mathbf{M} \quad \left\{ \begin{array}{l} \vdots \\ \text{IF (condition)} \\ \text{THEN Om = C} \\ \text{ELSE Om =2} \end{array} \right.$$

Figure 4

Module Which Transmits a Program Computation Error

Assume once again that the computation in P should set C equal to Ip+1 instead of Ip . If integration test data were chosen which never exercised the true branch of the condition in M , then the resulting value of Om would always be 2 and the error in the computation of P would go undetected by simply examining the output of the program.

These two examples have elements in common. In both cases there is an error in the code preceding the call to the validated module. The error causes one of the module's inputs to have an incorrect (not invalid) value; it is possible for the error in the module's input to <u>not</u> be reflected as an error in the module's output, since transmission of the error to an output may be dependent upon the particular path chosen through the module. It is therefore clear that, when integrating a previously validated module, one needs to know more than just that the module is correct. If information relevant to the module's internal structure is ignored, it is possible for both domain and computation errors in the integrated program to go undetected. Therefore it is natural to ask at this stage "What, in addition to knowing that the module is correct, will allow effective integration testing to be done?".

4. Detecting Integration Errors

Two approaches to answering the question posed at the end of the previous section are suggested by the examples presented in that section. Since our goal is to detect errors in the module's input, we could simply require that all input values to the module be output (along with the normal output of the calling program). This technique is not new, as programmers often print out values of intermediate/temporary variables. However it is often hard to know whether an intermediate program value is correct. More likely, the programmer is only interested in examining the final outputs of the (calling) program.

Therefore, we consider a second approach. It would appear that the chief problem presented in the previous section is that the module's output may be unaffected by the error in the calling program. This section therefore addresses the problem of determining how much integration testing need be done in the module in order to ensure that an error in the module's input results in an error in the module's output.

To examine this problem in greater detail we will first impose the following restrictions on the module.

1. restrict the module such that all inputs are assigned upon entry, and no inputs are reassigned later in the module.
2. restrict the output variables such that all output variables are assigned at the end of the module, i.e. on a given path after the first assignment to an output variable no more assignments may be made to program variables. In addition, no output variable of the module can be used as a reference within the module. (This implies that the order in which the output variables are assigned along a path does not matter.)
3. restrict all computations and predicates to be linear with respect to the module's inputs.

The first two restrictions serve primarily to simplify the notation which follows. Since any program can be written so that it conforms to these two restrictions, they are not fundamental limitations. The third restriction, though considerable, makes it easier to model the computation sequence along a path in the program. After the major results have been derived, we will discuss the relaxation of the third restriction.

Given these restrictions, a program can be modeled in the following manner.

Suppose the program contains m input variables I1,..., Im, n program variables P1,...,Pn, and ℓ output variables O1,...,Oℓ. We introduce an environment vector, $\overline{V}$, which contains the current value of all variables at some point of execution.

$$\text{i.e.} \quad \overline{V} = \begin{bmatrix} 1 \\ \text{value of } I1 \\ \vdots \\ \text{value of } Im \\ \text{value of } P1 \\ \vdots \\ \text{value of } Pn \\ \text{value of } O1 \\ \vdots \\ \text{value of } O\ell \end{bmatrix}$$

The 1 represents a position for constants. Its need will become apparent in what follows, as we describe computations and predicates of the module in terms of an environment.

Example 1 m=4 n=3 ℓ=2

```
         SUBROUTINE MODULE1(I1,I2,I3,I4,O1,O2)

  1.     P1=I1+I2

  2.     P2=2*I1-I3+I4

  3.     IF P1=0

  4.         THEN

  5.             P3=P2+3

  6.         ELSE

  7.             P3=P1+P2

  8.     ENDIF

  9.     O1=P3

 10.     O2=P1+P2+P3

 11.     RETURN

 12.     END
```

if $\{I1,\ I2,\ I3,\ I4\} = \{1,\ -1,\ 2,\ 3\}$

Then after executing statement 1, $\overline{V} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ 0 \\ \text{undef} \\ \text{undef} \\ \text{undef} \\ \text{undef} \end{pmatrix}$

At the end of the program $\overline{V} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ 0 \\ 3 \\ 6 \\ 6 \\ 9 \end{pmatrix}$

A computation in the program (which by assumption is linear) will be represented as a $1 + m + n + \ell$ by $1 + m + n + \ell$ matrix. Intuitively, a row of this matrix describes the effect of the computation on an individual input, program, or output variable.

For a single assignment statement, which assigns exactly one program or output variable, the matrix is just an identity matrix except in the row corresponding to the assigned variable. The entries in this row contain the coefficients of the input and program variables which appear on the right hand side of the assignment statement, placed in the appropriate columns.

<u>Example 2</u> Using the subroutine MODULE1 in example 1, the matrix C corresponding to statement 2 is

$$C(\text{stmt } 2) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

while that for statement 5 is

$$C(\text{stmt } 5) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

A sequence of computations along a given path can be represented as the product of the matrices corresponding to the individual assignment statements.

Example 3 The computation sequence corresponding to executing statements 1, 2 and 5 is

$$C(\text{stmt } 5) \times (C(\text{stmt } 2) \times C(\text{stmt } 1)) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Given this model, there are two separate ways in which an error in the input can be transmitted to an output of the module.

1. An error in the input can cause the correct path (i.e. the same path that would have been followed had there been no error) to be taken, but can result in an incorrect value being assigned to an output variable. With respect to the entire integrated program, such a situation can be viewed as a computation error. We will call such errors "integration time computation errors".

2. An error in an input can cause an incorrect path to be taken in the module which in turn results in a different computation being performed and hence an incorrect value in an output variable. With respect to the integrated program, this situation can be viewed as a domain error. We will call such errors "integration time domain errors".

5. Detection of Integration Time Computation Errors

We first examine the methods by which an input error to the module can be transmitted to an output variable along a given path (the computation error situation). Using the three restrictions introduced earlier, the results of the computations along a given path can be represented by

$$\overline{VF} = C(\theta\ell)\ldots.C(\theta 1)C(k)\ldots.C(1)\overline{V0}$$

where $C(k),\ldots.,C(1)$ represent the computations which assign program variables along the path, $C(\theta\ell),\ldots.,C(\theta 1)$ represent the assignments to the output variables, $\overline{V0}$ is the initial environment vector and $\overline{VF}$ is the final environment vector (the result of the computations).[2] The above expression can be condensed by matrix multiplication to

$$\overline{VF} = C(\theta)C(P)\overline{V0}$$

where $C(P) = C(k)\ldots.C(1)$ and $C(\theta) = C(\theta\ell)\ldots.C(\theta 1)$. This can be further reduced to

$$\overline{VF} = C\ \overline{V0}$$

where C is now a single $1 + m + n + \ell$ by $1 + m + n + \ell$ matrix which represents the results of all computations performed along the particular path.

---

2. We will adopt the convention that it is permissable to multiply undefined values by zero, resulting in a value of zero. Multiplication of undefined values by any nonzero quantity will result in a value which is undefined. (see Example 4)

The final expression for each program and output variable of C is in terms of the input variables and constant only, thus corresponding to performing a symbolic execution of the particular path.

<u>Example 4</u> Using the same subroutine as in Example 1, with $\{I1, I2, I3, I4\} = \{1, -1, 2, 3\}$, we have

$$\overline{VO} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ \text{undef} \\ \text{undef} \\ \text{undef} \\ \text{undef} \\ \text{undef} \end{pmatrix} \qquad C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 5 & 1 & -2 & 2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\overline{VF} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ 0 \\ 3 \\ 6 \\ 6 \\ 9 \end{pmatrix}$$

The question to be considered is "In what ways can an error in $\overline{VO}$ be transformed into an error in $\overline{VF}$?". If $\overline{VO}$ is incorrect then it can be represented as

$$\overline{VO} = \overline{VO}' + \overline{e} \qquad\qquad \overline{e} \neq \overline{0}$$

where $\overline{VO}'$ is the correct initial environment vector, and $\overline{e}$ is the initial error term in the environment vector. Since the initial error can only occur in the input variables, $\overline{e}$ is restricted to be 0 in the first position, and 0 in the last $n + \ell$ positions, and nonzero in at least one position from 2 to $m + 1$.

If this new expression is substituted into the expression for the path environment it yields

$$\overline{VF} = C\ (\ \overline{VO}' + \overline{e}\ )$$

or

$$\overline{VF} = C\ \overline{VO}' + C\ \overline{e}$$

Since the erroneous input follows the same path through the module, $C\ \overline{VO}'$ represents the "correct" final environment vector (i.e. the same C should have been applied to $\overline{VO}'$, the correct initial environment). Therefore the error is only detectable in the final environment vector if $C\overline{e} \neq \overline{0}$. However, this restriction isn't sufficient to <u>ensure</u> detection. This is because we are assuming that, as a result of executing a path through the module, only the "output variable part" of the final environment vector, and not the entire final environment vector, is available. Therefore, if the error in the input is to be detected, then

$$(C\overline{e})_{\text{elements } m+n+2 \text{ thru } m+n+\ell+1} \neq \overline{0} \quad (1)$$ [3]

---

3. As a notational convention, we will use subscripts to express a subset of elements of a vector or matrix. Thus, condition (1) can be written as

$$(C\ \overline{e})_{m+n+2,\ldots,m+n+\ell+1} \neq \overline{0}$$

Similarly, if we wished to describe the "upper left" submatrix of C containing the first y rows and z columns, we could write $C_{1,\ldots,y \times 1,\ldots,z}$

(Clearly, $C\,\bar{e}$ is never equal to $\bar{0}$ since $\bar{e} \neq \bar{0}$ and we have restricted the module so that it cannot reassign its input variables. Hence at least one of the elements in rows 2 through $m + 1$ must be nonzero.)

To better understand the meaning of this restriction that $C\,\bar{e}$ not be 0 in the last ℓ positions, let's examine the C matrix in greater detail.

C can be considered to have 9 submatrices of the following form.

$$C = \left[\begin{array}{c|c|c} C(1,1) & C(1,2) & C(1,3) \\ \hline C(2,1) & C(2,2) & C(2,3) \\ \hline C(3,1) & C(3,2) & C(3,3) \end{array}\right]$$

where:

$C(1,1)$ is an $m+1$ by $m+1$ matrix which describes how the inputs and constants are mapped onto the inputs and constants. (By our restrictions this must be equal to the identity matrix.)

$C(1,2)$ is an $m+1$ by n matrix which describes how program variables are mapped onto the inputs and constants. (This must be 0 by our restrictions.)

$C(1,3)$ is an $m+1$ by ℓ matrix which describes how outputs are mapped onto inputs and constants (also equal to 0).

C(2,1) is an n by $m+1$ matrix which describes how inputs and constants are mapped onto program variables. (This submatrix is unrestricted in form.)

C(2,2) is an n by n matrix which describes how program variables are mapped onto program variables. (This submatrix is 0 since C contains the results of a symbolic execution of the path and program variables in their final symbolic form are defined completely in terms of input variables and constants. The only possible exception is the row corresponding to a program variable which is not defined along the path. Such a row would have a 1 in the column corresponding to the program variable and zeros elsewhere.)

C(2,3) is an n by ℓ matrix which describes how outputs are mapped onto program variables (also equal to 0).

C(3,1) is an ℓ by $m+1$ matrix which describes how inputs and constants are mapped onto output variables. (This submatrix is unrestricted in form.)

C(3,2) is an ℓ by n matrix which describes how program variables are mapped onto outputs. (This submatrix must be 0 for the same reasons given for C(2,2) above.)

C(3,3) is an ℓ by ℓ matrix which describes how outputs are mapped onto outputs. (This must also be 0 , with the exception of "identity rows", as described in C(2,2), corresponding to output variables which are unassigned along the path.)

Example 5   Using the C matrix from Example 4, we have the following partition

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 5 & 1 & -2 & 2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

If we now return to the question of transforming an error in the input to an error in the output by looking at the breakdown, we note that the only interesting part of the C matrix is the submatrix C(3,1) which describes how inputs and the constant are mapped onto the output variables. The results of the computations along a path can now be described by

$$\overline{VF}_{1+m+n+1,\ldots,1+m+n+\ell} = C_{1+m+n+1,\ldots,1+m+n+\ell \times 1,\ldots,m+1} * \overline{VO}_{1,\ldots,m+1}$$

$$= C(3,1)\overline{VO}_{1,\ldots,m+1}$$

Again introducing the error term we get

$$\overline{VF}_{1+m+n+1,\ldots,1+m+n+\ell} = C(3,1)(\overline{VO}_{1,\ldots,m+1} + e_{1,\ldots,m+1})$$

or

$$\overline{VF}_{1+m+n+1,\ldots,1+m+n+\ell} = C(3,1)\overline{VO}_{1,\ldots,m+1} + C(3,1)\bar{e}_{1,\ldots,m+1}$$

so that the error can only be detected in the final environment vector for this path if $C(3,1)\bar{e}_{1,\ldots,m+1} \neq \bar{0}$.

We can further note that the constant position in $\bar{e}_{1,\ldots,m+1}$ is always zero. That is, we really have only m linearly independent "error directions", corresponding to the m input variables of the module (assuming no inherent relationships among these inputs). Therefore, the first column of $C(3,1)$ plays no part in determining whether $C(3,1)\bar{e}_{1,\ldots,m+1} \neq \bar{0}$. Defining $C'(3,1)$ to be $C(3,1)$ without the first column, we can now say that the input error can only be detected in the final environment vector for the path if $C'(3,1)\bar{e}_{1,\ldots,m+1} \neq \bar{0}$.

We therefore seek a feasible path for which it is possible to detect an input error in any of the m directions by examining the values of the output variables in the final environment vector. Such a path must "span the error space", so that all possible error directions are feasible along the path. Since the dimension of the error space and input space are the same, we will call such a path <u>input space spanning</u>.

We can now conclude

<u>Lemma 1</u>: Examination of the outputs of a module on any test exercising an input space spanning path for which

$$\bar{e}_{2,\ldots,m+1} \neq \bar{0} \Rightarrow C'(3,1)\,\bar{e}_{2,\ldots,m+1} \neq \bar{0} \quad (2)$$

is sufficient to detect any integration time computation error affecting the module.

From linear algebra, we know that

<u>Lemma 2</u>: A path satisfies condition (2) if there exists an $m \times m$ matrix M consisting of m rows of $C'(3,1)$ such that $|M| \neq 0$.

Intuitively a path satisfying the conditions of Lemma 2 transforms the inputs to outputs in such a way that there are m linearly independent functions of the inputs computed on that path.

Unfortunately, the results may not be very helpful since

a) there is no guarantee that such a path exists (in particular, if $\ell < m$, as in example 1, no such path can exist), and

b) even if there is such a path, finding it, or even showing its existence, may require a substantial amount of computation. For example, for each path in the module, there may be $\binom{\ell}{m}$ matrices whose determinants must be checked.

<u>Example 6</u> Using the program of Example 1, we note that neither path is sensitive to all possible error directions. The "then" path, analyzed in Example 5, gives

$C'(3,1) = \begin{pmatrix} 2 & 0 & -1 & 1 \\ 5 & 1 & -2 & 2 \end{pmatrix}$ which clearly contains no 4×4 matrix having nonzero determinant. It is easy to see that this path cannot detect an error where both I3 and I4 are incorrect by the same amount k . In such a case

01 = 3+2*I1 - (I3+k) + (I4+k) = 3+2*I1 - I3 + I4

02 = 3+5*I1 + I2 - 2*(I3+k) + (I4+k) = 3+5*I1 + I2 - 2*I3 + 2*I4

which are the same results as would have been produced with correct values of I3 and I4 .

The reader can verify that the "else" path is also insensitive to certain error directions.

- - - - - - - - - -

However, it is not necessary for a path satisfying Lemma 2 to exist. Consider an $m \times m$ matrix M' constructed from various rows in $C'(3,1)$ matrices taken from different input space spanning paths through the module. That is, M' consists of the results of some subset of output variables along some set of paths through the module. If $|M'| \neq 0$, then $e_{2,\ldots,m+1} \neq 0 \Rightarrow M'e_{2,\ldots,m+1} \neq 0$. Intuitively this means that we only need to find m linearly independent functions of the inputs computed somewhere in the module, even if we need to select several paths to find them. Of course, it is entirely possible that the same output variable will need to be examined on more than one path in order to obtain this set of functions. We can now conclude

Lemma 3: Examination of the appropriate output variables from any test exercising those paths corresponding to the construction of M' as defined above is sufficient to detect any integration time computation error affecting the module.

Example 7

```
SUBROUTINE MODULE2(I1, I2, 01, 02)
  IF  I1 = 0
  THEN
      01 = 2*I1+1
      02 = I1+2
  ELSE
      IF  I2 = 0
      THEN
          01 = I2+1
          02 = 2*I2+2
      ELSE
          01 = I1+I2
          02 = 3
      ENDIF
  ENDIF
```

No path is by itself sensitive to every error direction. However any two paths are sensitive to all error directions with the stipulation that if the "else-else" path is selected, 01 must be examined. The relevant C'(3,1) matrices for this example are

$$\begin{pmatrix} 2 & 0 \\ 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 \\ 0 & 2 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

The following theorem is an immediate consequence of Lemma 3.

Theorem 1: In order to detect any integration time computation error affecting a previously validated module, it is sufficient to test at most m input space spanning paths from the module, chosen so as to guarantee the existence of M' defined above.

Proof: A path need not be chosen for inclusion in the integration test set unless it is sensitive to some error direction that no other paths in the test set are sensitive to. The result then follows from the fact that there are only m error directions to begin with.

In many cases fewer than m paths will be required since one or more paths may contribute multiple rows to M'. If the particular module being examined has at least as many outputs as inputs it is possible that a single path will be sufficient to pass all possible errors in the input to errors in the output.

Theorem 1 still leaves the following open problems:

a) How does one find the proper C'(3,1) rows in order to build the $m \times m$ non-singular matrix? For a given program with p paths, ℓ outputs, and m inputs there are

$$\binom{\ell \times p}{m}$$

different candidates for M'. Clearly some path selection method must be used to reduce this large number of candidates.

b) The possibility exists that no $m \times m$ non-singular matrix exists (i.e. all paths through the module are "blind" to one or more error directions). Is there some method to discover this without looking at all candidates for the $m \times m$ non-singular matrix? (The reader should verify that the program of Example 1 is in fact completely blind to the error direction discussed in Example 6.) We will call those integration time computation errors, which are in directions which all paths are not blind to, "detectable integration time computation errors".

c) The requirement that paths be "input space spanning" needs further examination. Equality predicates, or combinations of predicates which imply an equality condition (such as $A \leq B$ and $A \geq B$), along a given path make it impossible for certain types of integration errors to be detected since the correct and erroneous inputs can never follow the same path. However the computations along that path might still satisfy the conditions of Lemma 2.

For example, consider a module with inputs I and J, outputs O_1 and O_2, and a program segment

```
     .
     .
     .
IF I = 1
THEN
     O1 = I + J
     O2 = 2 + J
ELSE
     .
     .
     .
```

Examination of just the computations assigning O_1 and O_2 might lead us to conclude that this "then" path is sufficient to detect any error in I and J (since $\begin{vmatrix} 1 & 1 \\ 0 & 1 \end{vmatrix} \neq 0$), while in reality it's only powerful enough to detect errors in the input when both I is equal to 1 and I <u>should be</u> equal to 1. In effect, there are no input errors in I which cause this path to be followed when it is correct to follow this path. We should, therefore, restate Lemmas 1 and 2 to say that such a path is powerful enough to detect any input error for which the path is still the correct one to be followed. All other input errors really manifest themselves as integration time <u>domain</u> errors from the point of view of this path. Integration time domain errors are studied in the following section.

We conclude this section by noting that, since there are only m linearly independent error directions to be covered,

<u>Theorem 2</u>: A set of at most m "beneficial" paths (where a path is added to the set iff it covers a new error direction) will suffice to transmit any detectable integration time computation error in an input to a correct module to some output of the module.

If the paths are input space spanning, we need only compute the matrix information discussed previously. If not, we must further ensure that an error direction which we intend a path to cover is in fact feasible along that path.

6. <u>Detection of Integration Time Domain Errors</u>

We now address the second type of integration error presented in Section 4, that of an error in the calling program which causes the (incorrect) input to the module to follow a different path. This situation comes about when the error causes one of the module's predicates to have an interpretation which is nonequivalent to that which correct input would have produced.

The model of Section 4 needs to be expanded slightly in order to incorporate the idea of a predicate interpretation. Given a particular predicate T in the module that is under examination, and a path leading to that predicate, that predicate interpretation can be modeled as

$$0 \ \text{.relop.} \ \overline{T}^{t} \, C \, \overline{VO}$$

where	$\overline{VO}$	represents the initial environment vector (as before)
	C	represents the results of the computations in the module along the path leading to the predicate.
	$\overline{T}$	represents the predicate which when applied to $C\,\overline{VO}$ yields a scalar which is compared to zero to determine whether to branch or not. i.e., the elements of $\overline{T}$ contain the coefficients of the constant and variables used in the predicate. (The transpose of $\overline{T}$ is required in this scalar product since $\overline{T}$ is a column vector.)
	.relop.	is any relational operator which determines the type of comparison being made.

<u>Example 8</u> Using the module of Example 1 again, the predicate P1 $\geq$ 0 can be represented as

$$\bar{T} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Since there is only one path leading to this predicate,

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\overline{VO} = \begin{pmatrix} 1 \\ \text{value of I1} \\ \text{value of I2} \\ \text{value of I3} \\ \text{value of I4} \\ \text{undefined} \\ \text{undefined} \\ \text{undefined} \\ \text{undefined} \\ \text{undefined} \end{pmatrix}$$

Thus the interpretation of the predicate is

$$0 \leq \bar{T}^t C \, \overline{VO}$$

or

$$0 \leq I1 + I2$$

which, when evaluated for $\{I1, I2, I3, I4\} = \{1, -1, 2, 3\}$, yields

$$0 \leq 0 \quad \text{or} \quad \text{TRUE}.$$

- - - - - - - - - - -

In order for an error in the calling program to produce nonequivalent predicate interpretation along some given path in the correct module, it is necessary for the following condition to hold.

$$\nexists k \neq 0 \ni$$
$$\bar{e} \neq \bar{0} \Longrightarrow \bar{T}^{t} C \, (\overline{VO} + \bar{e}) \neq k \, \bar{T}^{t} C \, \overline{VO} \qquad (3)$$

where $\bar{e}$ represents the error vector. That is, the erroneous interpretation of the predicate should not be a multiple of the correct interpretation, for otherwise the path taken by any input reaching this predicate along the given path will not be altered.

<u>Example 9</u>

Using the subroutine of Example 1 once again, we consider the predicate $P1 \geq 0$, whose interpretation is $I1 + I2 \geq 0$. If an error in the calling module causes I1 and I2 to be modified (to I1' and I2') in such a way that $I1' + I2' = k(I1 + I2)$ for some $k \neq 0$, then the interpretation of this predicate (i.e. $k(I1 + I2) \geq 0$) is equivalent to the original. As an example of such a situation, suppose the calling program had inputs X, Y, and Z and further suppose that computations in the calling program <u>should have</u> set

```
I1 - X + 2*Y
I2 = 2*Z
```

but erroneously set

```
I1 = 2*X + Z
I2 = 3*Z + 4*Y
```

Then, in terms of the inputs to the calling module, the interpretation of $P1 \geq 0$ should have been $X + 2*Y + 2*Z \geq 0$, but instead is $2*X + 4*Y + 4*Z = 2*(X + 2*Y + 2*Z) \geq 0$. Both the correct and incorrect interpretations evaluate identically for any triplet (X, Y, Z).

As in the previous section, we must be cognizant of the effect non input space spanning paths have on the ability of a predicate to undergo a change in interpretation.

Example 10 The predicate $P1 \geq 0$ from Example 1, whose interpretation is $I1 + I2 \geq 0$, appears to be sensitive to changes in the input I1 as long as its new interpretation is not $\lambda(I1 + I2) \geq 0$. However, suppose that this predicate appeared along a path in the module previously constrained by a predicate $I1 = 1$. Since $P1 \geq 0$ cannot even be reached if $I1 \neq 1$, it can be said to have no interpretation under these conditions. Hence, its interpretation cannot be affected by a change in I1, so that P1 is no longer helpful in identifying any errors in I1.

Let us now examine the C matrix in greater detail. The C matrix can be considered as 9 submatrices (as before). The only submatrix of interest this time is C(2,1) (the submatrix which defines how inputs are mapped onto program variables), as only input and program variables can be used in the predicate. Likewise, the only interesting positions of $\overline{T}$ are the first $m + n + 1$ positions (all the rest must be 0). Therefore if condition (3) is grouped in the following way.

$$\nexists k \neq 0 \ni$$
$$e \neq 0 \Rightarrow (\bar{T}^t C)\ (\overline{VO} + \bar{e}) \neq k (\bar{T}^t C)\ \overline{VO}$$

then the expression $(T^t C)$ is the predicate interpretation vector of which only the first $m + 1$ positions can be non-zero. These positions represent the manner in which input variables are mapped onto the predicate scalar. Hence we can expand condition (3) as follows.

$$\nexists k \neq 0 \ni \bar{e} \neq \bar{0} \Rightarrow$$
$$((\bar{T}^t C)_{1,\dots,m+1} \overline{VO}_{1,\dots,m+1}) + ((\bar{T}^t C)_{1,\dots,m+1} \bar{e}_{1,\dots,m+1}) \neq$$
$$k((\bar{T}^t C)_{1,\dots,m+1} \overline{V\Theta}_{1,\dots,m+1})$$

where the subscripts indicate that just the first $m + 1$ positons of each vector are being used. As a result, we can conclude

<u>Lemma 4</u>: Under restrictions 1-3 of Section 4, in order to ensure that an error in the calling program produces a nonequivalent predicate interpretation along some given path in the correct module, it is sufficient that there exist a predicate T in the module and an input space spanning path leading to T satisfying

$$\nexists k \ni$$
$$\bar{e}_{1,\dots,m+1} \neq \bar{0} \Rightarrow (\bar{T}^t C)_{1,\dots,m+1}\ \bar{e}_{1,\dots,m+1} \neq k (\bar{T}^t C)_{1,\dots,m+1} \overline{VO}_{1,\dots,m+1}$$

We are now faced with the problem of how to detect such changes in a predicate interpretation. Fortunately, the Domain Testing Strategy [White, 80] provides an answer to this question. By selecting a small number of test points at or near the border of the input space corresponding to the predicate interpretation, we can guarantee that essentially all changes (up to a parameter ϵ) in interpretation are detected. In order to use this result, it is necessary to assume that adjacent regions of the module's input space compute different functions which do not give "coincidentally identical" results on the set of test data chosen. With this in mind, we have

<u>Lemma 5</u>: By retesting a predicate statisfying Lemma 4 using the Domain Testing Strategy, it is possible to detect any integration time domain error which results in a change of magnitude greater than ε in the predicate interpretation.

As in the previous section, we note that

a) there is in general no predicate satisfying the conditions of Lemma 4, and

b) since $e_1 = 0$, there are really only m linearly independent error directions.

However, if we can select a set of m predicate interpretations $\left\{[(\bar{T}^tC)_{2,\ldots,m+1}]_i\right\}_{i=1}^{m}$ from input space spanning paths such that the $m \times m$ matrix M'' whose i^{th} row consists of $[(\bar{T}^tC)_{2,\ldots,m+1}]_i$ has a non-zero determinant, then

$$\nexists\, k \ni$$

$$\bar{e}_{2,\ldots,m+1} \neq \bar{0} \Longrightarrow [(\bar{T}^tC)_{2,\ldots,m+1}]_i\ \bar{e}_{2,\ldots,m+1} \neq k\,[(T^tC)_{2,\ldots,m+1}]_i\ \overline{VO}_{2,\ldots,m+1}$$

for all $i = 1,\ldots,m$

We can therefore conclude

<u>Theorem 3</u>: By retesting, using the Domain Testing Strategy, a set of input space spanning paths through a correct module such that the set contains m predicate interpretations satisfying the condition on M'' defined above, it is possible to detect any integration time domain error in the module of Domain Testing consequence $> \varepsilon$.

As was the case in the previous section, we have no assurance that we can ever find such a set of predicate interpretations, nor do we have a computationally efficient method of finding them if they exist.

We also note, as before, that the "input space spanning" requirement appears to impose additional requirements on the paths we can choose. However, as before, a set of at most m "helpful" paths, each of which may itself be insensitive to certain error directions, but which covers an error direction that the other paths don't, will suffice to detect any "detectable" integration time domain error.

7. The Linearity Requirements

The results presented in the previous sections were developed under the very restrictive assumption that all computations and predicates were linear with respect to the module's inputs. However, that assumption was not used in its entirety. For example, we can apply the results for detecting integration time computation errors as long as we can find a set of paths which produce enough linear functions of the module's inputs so that the nonsingular matrix M' can be produced. It is certainly more realistic to assume that some functions computed along some paths in the module are linear.

For integration time domain errors, it is also not necessary to have all computations and predicates be linear. Rather, we require that enough paths containing linear predicate interpretations be found so that the matrix M'' can be constructed. Note that this restriction is even weaker than that of a "linearly domained" module [Zeil, 81] , where every predicate is required to have a linear interpretation.

Final Remarks

We have shown that, if an error exists in a computation preceding a call to a previously validated module, and if that error results in an error in the module's input, it is theoretically possible to detect this condition without having to deal with the entire path complexity of the module. In fact, the maximum number of paths that need be retested depends only on the dimensionality of the module's input space. Sufficient conditions on this set of paths have also been presented.

It remains to be shown whether computationally efficient methods exist for finding the right set of paths to be tested. From the development of the results, "input space spanning" paths seem to offer the most promise. This suggests that choosing false branches of equality predicates and strict inequality branches of other predicates may be a useful heuristic in attempting to cover all error directions.

We have noted the possibility that no set of paths satisfying the retesting conditions exists. One may be tempted to argue that the likelihood of such a phenomenon is small, since the set of functions and predicates constraining the error directions has measure zero with respect to the error space under the standard "equal likelihood" assumptions. However, our intuition tells us that in real programs the equal likelihood assumption does not apply.

We can further note that even if we can determine the existence of a set of paths which, when retested, will be sure to transmit any input error to the output of the module, we must now face the problem of generating test data for the calling program which will exercise these paths. Our ability to generate such test data may well be constrained by the structure of the calling program. That is, we may only be able to generate data which exercise the key paths in the correct module if we follow certain paths in the calling program. This may inhibit our ability to thoroughly test certain parts of the calling program. Experiments performed on real programs should provide useful answers as to the severity of these problems.

We have approached the problem of integration testing in this paper from a "bottom up" point of view, in that we were concerned with the integration of a previously tested module into a higher level software unit. The resulting integration testing strategy involves selecting carefully chosen paths in the (completely developed) module for retesting. However, the results of this investigation suggest that one might equally well have explored the problem from the "top down" point of view. That is, we might explore the idea of using the notion of "linearly independent" "input space spanning" paths in stub development, for it is at this point in top-down development that we are really concerned with error conditions in the higher level software unit.

Finally, we note that errors in the higner level unit which result in incorrect module input values constitute only one class of integration problems. For example, we might have errors (in the calling module) in the code following the call to the correct module. The ability to detect these errors, however, may depend on which path through the module was followed, since the program statement in error may involve output variables from the correct module. It remains to be shown whether the paths to be selected to catch input errors to the module are in any way related to the paths required to detect these errors.

Despite these problems, formidable as they are, it is comforting to have the intuitively appealing result that, from a theoretical standpoint, it is only necessary to retest a small number of possible paths through a correct module in order to detect certain integration errors. It is our hope that these results can form the basis of a more unified and systematic approach to integration testing, so that some form of (partial) certification of a software system may be possible.

References

[Howden, 76] Howden, W. "Reliability of the Path Analysis Testing Strategy", IEEE Trans. on Software Eng., SE-2, 3, September, 1976, p. 208-214.

[Howden, 79] Howden, W., "Effectiveness of Software Validation Methods", Infotech State of the Art Report on Software Testing, Vol. 2, Infotech International, 1979, p 131-146.

[Parnas, 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, 15, 12, December 1972, p. 1053-1058.

[Stevens, 74] Stevens, W., Myers, G. and Constantine, L., "Structured Design", IBM Systems Journal, No. 2, 1974, p. 115-139.

[Westley, 79] Westley, A., ed., Infotech State of the Art Report on Software Testing, Vol. 1, Infotech International, 1979.

[White, 80] White, L. and Cohen, E., "A Domain Strategy for Computer Program Testing", IEEE Trans. on Software Eng., SE-6, 3, May 1980, p. 247-257.

[Yourdon, 79] Yourdon, E. and Constantine, L., Structured Design, Prentice Hall, Englewood Cliffs, N. J., 1979.

[Zeil, 81] Zeil, S. and White, L., "Sufficient Test Sets for Path Analysis Testing Strategies", Proc. 5th Int'l. Conf. on Software Engineering, San Diego, CA, March 1981, p. 184-191.

SECURITY CLASSIFICATION OF THIS PAGE (When Data E

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 81 -0578 — 2. GOVT ACCESSION NO. AD-A102 477	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN APPROACH TO RELIABLE INTEGRATION TESTING	5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
	6. PERFORMING ORG. REPORT NUMBER OSU-CISRC-TR-81-5 ✓
7. AUTHOR(s) Allen Haley and Stuart Zweben	8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0152
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer and Information Science Research Center Ohio State University Columbus OH 43210	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB DC 20332	12. REPORT DATE MAY 81
	13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A testing strategy which involves integrating a previously validated module into a software system is described. It is shown that, when doing the integration testing, it is not enough to treat the previously validated module as a "black box", for otherwise certain integration errors may go undetected. For example, an error in the calling program may cause an error in the module's paths through the module. The results indicate that such errors can be detected by the module by retesting a set of paths whose cardinality depends only on the dimensionality of the module's input space, rather than on the module's path complexity.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DATE
ILME